Trofimov, Fedor (Grupo 31)
Práctica PRO2 Primavera 2024: Justificación de código

comerciar

Código fuente:

```
// Pre:  Ciudad implícita es diferente a Ciudad c
// Post: Ciudad implícita le ha dado todos los productos que le sobran a Ciudad c hasta
         alcanzar, si es posible los que la Ciudad c necesita, y viceversa.

void Ciudad::comerciar_con(Ciudad& c, const vector<pair<int,int>>& id2infoprod)
{
    map<int,Producto>::iterator it1 = inventario.begin();
    map<int,Producto>::iterator it2 = c.inventario.begin();
    int cambio_peso = 0;
    int cambio_volumen = 0;

    while(it1 != inventario.end() and it2 != c.inventario.end())
    {
        // 3 casos:
        // Ambos it apuntan a productos con ids iguales:  intentar comerciar el producto
        // It 1 apunta a un producto con id mas grande:   aumentar iterador 2
        // It 2 apunta a un producto con id mas grande:   aumentar iterador 1

        if (it1->first == it2->first)
        {

            int c1_exceso = it1->second. Producto::obtener_exceso();
            int c2_exceso = it2->second. Producto::obtener_exceso();

            // 2 casos:
            //     - Ambas ciudades quieren o les sobra el producto:      no hacer nada
            //     - Una ciudad quiere el producto y a la otra le sobra:  hacer comercio

            if (c1_exceso < 0 and c2_exceso > 0) // Ciudad impl. quiere y a Ciudad c le sobra
            {
                int cantidad_comerciada = min(c2_exceso, -c1_exceso);

                it1->second += cantidad_comerciada;
                it2->second += -cantidad_comerciada;

                cambio_peso += cantidad_comerciada * id2infoprod[it1->first].first;
                cambio_volumen += cantidad_comerciada * id2infoprod[it1->first].second;
            }

            if (c2_exceso < 0 and c1_exceso > 0) // Ciudad c quiere y a Ciudad impl. le sobra
            {
                int cantidad_comerciada = min(c1_exceso, -c2_exceso);

                it1->second += -cantidad_comerciada;
                it2->second += cantidad_comerciada;

                cambio_peso -= cantidad_comerciada * id2infoprod[it1->first].first;
                cambio_volumen -= cantidad_comerciada * id2infoprod[it1->first].second;
            }
```

```
            it1++;
            it2++;
        }
        else if (it1->first > it2->first) it2++;
        else it1++;
    }

    peso_total += cambio_peso;
    volumen_total += cambio_volumen;
    c.peso_total -= cambio_peso;
    c.volumen_total -= cambio_volumen;
}
```

Justificación:

Precondición
Las dos ciudades son distintas

Postcondición
Las dos ciudades han comerciado los productos que tienen en común en sus inventarios.

Inicializaciones
Se han declarado dos iteradores sobre los dos inventarios de la ciudad. Estos iteradores de tipo `map` tienen un miembro `first` que indica el identificador de producto (`int`), y miembro `second` que es la instancia de clase Producto. Sabemos que los mapas de llave `int` están ordenados por defecto en orden ascendiente.

Invariante del bucle `while()`
Usando "id1" para referirme al identificador del producto al cual apunta it1 (`it1->first`), y similarmente "id2"

$$id1 < id2 \ \vee id1 > id2 \vee id1 = id2 \vee it1 = inv.end() \vee it2 = c.inv.end()$$

También, se han comerciado todos los productos comunes de los inventarios que tienen id menor que id1 e id2.

¿Precondición y Inicializaciones implican Invariante?
Claramente, después de inicializar los iteradores, un id va a ser mayor que el otro, van a ser iguales, o uno de ellos apunta a `.end()` (en el caso de que el inventario es vacío)

Condición de entrada del bucle (B)
El bucle continuará mientras ambos inventarios tengan productos que no se han tratado. Es decir, mientras que uno de los iteradores no llegue al final del `map`

Condición de salida del bucle (NOT B)
Cuando uno de los dos iteradores llega al final. Esto indica que ya se han recorrido todos los productos de una ciudad y por lo tanto no se puede seguir comerciando.

¿Invariante y B implican Invariante?
Analizando el bucle hay 3 casos:

- `id1==id2`

En este caso, el producto está en ambos inventarios, y por lo tanto se puede intentar comerciar. La función analiza la cantidad a comerciar, actualiza los inventarios de las ciudades adecuadamente y guarda el cambio total del peso y volumen de las ciudades. Se aumentan ambos iteradores.

- `id1 < id2`

En este caso se aumenta it1. Como los mapas están ordenados, este aumento podría producir el primer caso en la siguiente iteración del bucle.

- `id1 > id2`

Similar al caso 2, pero con el iterador del segundo inventario.

Nos damos cuenta de que al final de cada caso se aumentan ambos o solo un iterador. Es decir, `id1` o `id2` o ambos aumentarán o algun iterador llegara al final (`.end()`)
También nos damos cuenta de que todos los productos coincidentes con identificadores menores a `id1` y `id2` se han comerciado en iteraciones anteriores del bucle. Todo esto, entonces, claramente implica el invariante.

¿Invariante y no B implican Post?
Por una parte, !B quiere decir que ya no quedan productos comunes que comerciar. Por otra parte, el invariante nos indica que se han comerciado todos los productos con identificadores coincidentes menores a id1 y id2. Por lo tanto se puede concluir que las dos ciudades han comerciado todos los productos que tienen en común en sus inventarios, qué es exactamente el Post.

¿El bucle acaba?
Para ver que el bucle acaba procedo a definir una función de hito que es estrictamente decreciente. Usando pseudocódigo:

$$f = inv.size - it1 + c.inv.size - it2 + 1$$

Nos damos cuenta que con cada iteración al menos uno de los dos iteradores se aumenta, lo que demuestra que *f* es estrictamente decreciente. Entonces, el bucle acabará, cuando *f* llegue a 1 o cuando *inv.size == it1* o *c.inv.size == it2*. También nos damos cuenta que con que se cumpla el Pre, la función *f* siempre dará una imagen que pertenece a los números naturales.

hacer_viaje

Código fuente:

```
// Pre:  True
// Post: Los nodos del árbol son instancias del struct auxiliar InfoViaje.
         El tamaño de arbol auxiliar es menor o igual al tamaño de la cuenca

BinTree<Barco::InfoViaje> Barco::hacer_viaje_arbol_aux
(const BinTree<string>& cuenca, const map<string, Ciudad>& nombre2ciudad, int
potencial_comprar, int potencial_vender) const
{
    Barco::InfoViaje v = Barco::InfoViaje();
    // Casos base:
    //     - La cuenca está vacía
    //     - No queda más potencial de vender/comprar

    if (cuenca.empty())                                      return BinTree<InfoViaje>();
    if (potencial_comprar <= 0 and potencial_vender <= 0)    return BinTree<InfoViaje>();

    // Caso recursivo:
        // Cálculo de la cantidad que se podrá comprar/vender en la Ciudad del nodo con las
           'unidades restantes' en el barco (potencial_comprar/vender).
        // Las ciudades no se ven modificadas dentro de estos cálculos.

    Ciudad c = nombre2ciudad.at(cuenca.value());

    v.compra = 0;
    if (c.existe_producto_en_inventario(comprar_id))
    {
        int exceso = c.exceso_de_producto_en_inventario(comprar_id);
        if (exceso > 0)
        {
            if (exceso <= potencial_comprar) v.compra = exceso;
            else v.compra = potencial_comprar;
        }
    }

    v.venta = 0;
    if (c.existe_producto_en_inventario(vender_id))
    {
        int exceso = c.exceso_de_producto_en_inventario(vender_id);
        if (exceso < 0)
        {
            exceso = abs(exceso);
            if (exceso <= potencial_vender) v.venta = exceso;
            else v.venta = potencial_vender;
        }
    }

    auto l = hacer_viaje_arbol_aux(cuenca.left() , nombre2ciudad, potencial_comprar-v.compra,
             potencial_vender-v.venta);
    auto r = hacer_viaje_arbol_aux(cuenca.right(), nombre2ciudad, potencial_comprar-v.compra,
             potencial_vender-v.venta);

        /* Cálculo de los atributos del nodo (InfoViaje)
           4 casos:
```

```
            - Potencial izq. == Potencial der. == 0
                Si no se compra/vende nada en el nodo, 'altura' es 0. En caso contrario
                'altura' es 1.
                'compra_acumulada' es la compra del nodo.
                (Similarmente con 'venta_acumulada')

            - Potencial izq. >  Potencial der.
                'altura' es la altura del subarbol izq. + 1
                'compra_acumulada' es la suma de la compra acumulada del subarbol izq. y la
                compra del nodo.
                (Similarmente con 'venta_acumulada')

            - Potencial izq. <  Potencial der.
                'altura' es la altura del subarbol der. + 1
                'compra_acumulada' es la suma de la compra acumulada del subarbol der. y la
                compra del nodo.
                (Similarmente con 'venta_acumulada')

            - Potencial izq. == Potencial der. != 0
                'altura' del nodo es la altura mínima de los dos subarboles + 1
                'compra_acumulada' es la suma de la compra acumulada del subarbol con menor
                altura y la compra del nodo.
                (Similarmente con 'venta_acumulada')
    */

int lpot = 0;
if (!l.empty()) lpot = l.value(). InfoViaje::potencial();

int rpot = 0;
if (!r.empty()) rpot = r.value(). InfoViaje::potencial();

if (lpot == 0 and rpot == 0)
{
    v.altura = 0;
    v.compra_acumulada = v.compra;
    v.venta_acumulada = v.venta;
    if (v.potencial() > 0) v.altura = 1;
}
else if (lpot > rpot)
{
    v.altura = l.value().altura + 1;
    v.compra_acumulada = l.value(). InfoViaje::compra_acumulada + v.compra;
    v.venta_acumulada  = l.value(). InfoViaje::venta_acumulada + v.venta;
}
else if (lpot < rpot)
{
    v.altura = r.value().altura + 1;
    v.compra_acumulada = r.value(). InfoViaje::compra_acumulada + v.compra;
    v.venta_acumulada  = r.value(). InfoViaje::venta_acumulada + v.venta;
}
else
{
    v.altura = min(l.value().altura, r.value().altura) + 1;
    if (l.value().altura > r.value().altura)
    {
        v.compra_acumulada = r.value(). InfoViaje::compra_acumulada + v.compra;
        v.venta_acumulada  = r.value(). InfoViaje::venta_acumulada + v.venta;
```

```
        }
        else if (l.value().altura <= r.value().altura)
        {
            v.compra_acumulada = l.value(). InfoViaje::compra_acumulada + v.compra;
            v.venta_acumulada  = l.value(). InfoViaje::venta_acumulada + v.venta;
        }
    }

    return BinTree<InfoViaje>(v, l, r);
}
```

Justificación:

Precondición
Cierto

Postcondición
Los nodos del árbol son instancias del struct auxiliar InfoViaje y el tamaño de árbol auxiliar es menor o igual al tamaño de la cuenca.

Casos base
Caso 1: Cuenca vacía.
Este caso ocurre cuando el árbol de la cuenca está vacío. Como una cuenca vacía no representa una ciudad, el barco no puede intentar comerciar.

> **¿Implica Post?**
> Si la cuenca está vacía entonces la función devuelve un árbol de struct InfoViaje vacío. Como la cuenca y el árbol devuelto son ambos vacíos, se puede decir que el tamaño del árbol auxiliar es igual al tamaño de la cuenca, y por lo tanto el Post se cumple.

Caso 2: Los productos a comprar/vender se han acabado.
Este caso ocurre cuando ambos parámetros `potencial_comprar, potencial_vender` son inferiores o igual a cero. Esto en un caso base porque el barco no puede comerciar con ninguna ciudad más, ya que no le quedan productos a comprar o vender.

> **¿Implica Post?**
> En el caso de que se hayan acabado los productos a comprar/vender, la función también devuelve un árbol de struct InfoViaje vacío. En este caso, el árbol auxiliar tendrá un tamaño menor al tamaño de la cuenca, por lo que también se cumple el Post.

Caso recursivo
La manera en que se trata el árbol en el caso recursivo es una combinación de preorden y postorden. Primero se calcula la cantidad que el barco puede comprar y vender sus productos en la ciudad del nodo de la cuenca. Esto se hace considerando los parámetros `potencial_vender` y `potencial_comprar`.

Como se ha calculado la cantidad que el barco puede comprar y vender en la raíz en preorden, usaremos el nuevo potencial `potencial_comprar = POTENCIAL_COMPRAR - compra` (y similarmente con la venta) para las llamadas recursivas. Entonces, para el árbol auxiliar izquierdo llamamos recursivamente a la misma función con parámetro `cuenca.left()` y para el árbol auxiliar derecho llamamos recursivamente a la misma función con parámetro `cuenca.right()`.

Asi entonces, las llamadas recursivas tendrán en cuenta la cantidad que el barco ya ha comprado y vendido anteriormente (así acercándonos al caso base 2)

Después de las llamadas recursivas la función procede a calcular el resto de información del struct InfoViaje pero ahora ya en postorden.

Accede a los subárboles auxiliares ya calculados recursivamente y determina el “potencial” de la raíz. Es decir, el potencial de la raíz será el potencial máximo de los subárboles más la compra y venta hecha en esta ciudad. También se calcula la “altura” del nodo. Esto indicará la longitud del camino más provechoso y por lo tanto la altura de la raíz se establece a la altura del subárbol con mayor potencial más 1. Si los potenciales son iguales, se establece a la altura del subárbol con la altura mínima más 1 (ya que nos interesa el camino más corto).

Un caso particular es sí ambos subárboles tienen potencial nulo. Esto puede pasar si por ambos caminos el barco llega a las hojas del árbol de la cuenca sin vender ni comprar nada más. En este caso, el potencial de la raíz será la propia compra/venta en la ciudad y la altura será 1 si se ha vendido o comprado algo o cero si no.

Finalmente, el caso recursivo devuelve un árbol de tipo InfoViaje, incluyendo los subárboles calculados recursivamente. Este árbol tiene tamaño igual o menor que la cuenca, y por lo tanto se cumple el Post.

Decrecimiento
Nos podemos dar cuenta que cada llamada recursiva nos acerca a ambos casos base. Por un lado, decrece la altura del árbol (que en algún punto tendrá que llegar a ser un árbol vacío) y por el otro lado decrece el potencial a comprar y vender (que posiblemente nos lleve al caso base antes de llegar a las hojas del árbol de la cuenca). Por lo tanto las llamadas recursivas acabarán siempre.